Enable Simultaneous DNN Services Based on Deterministic Operator Overlap and Precise Latency Prediction

Weihao Cui*, Han Zhao*, Quan Chen*◇, Ningxin Zheng†, Jingwen Leng*◇, Jieru Zhao*◇, Zhuo Song‡, Tao Ma‡, Yong Yang‡, Chao Li*◇, Minyi Guo*◇
{weihao,zhaohan_miven,chen-quan,zhao-jieru,chaol,myguo}@sjtu.edu.cn
Ningxin.Zheng@microsoft.com,{songzhuo.sz,boyu.mt,zhiche.yy}@alibaba-inc.com
*Shanghai Jiao Tong University,◇Shanghai Qi Zhi Institute,†Microsoft Research Asia, ‡ Alibaba Cloud

ABSTRACT

While user-facing services experience diurnal load patterns, co-locating services improve hardware utilization. Prior work on co-locating services on GPUs run queries sequentially, as the latencies of the queries are neither stable nor predictable when running simultaneously. The input sensitiveness and the non-deterministic operator overlap are two primary factors of the latency unpredictability. Hence, We propose **Abacus**, a runtime system that runs multiple services simultaneously. Abacus enables deterministic operator overlap to enforce latency predictability. Abacus composes of an *overlap-aware latency predictor*, a *headroom-based query controller*, and *segmental model executors*. The predictor predicts the latencies of the deterministic operator overlap. The controller determines the appropriate operator overlap for the QoS guarantee of all the services. The executors run the operators as needed to support the deterministic operator overlap. Our evaluation shows that Abacus reduces 51.3% of the QoS violation and improves the throughput by 29.8% on average compared with state-of-the-art solutions.

KEYWORDS

DNN services, QoS, latency prediction, GPU, Co-location

ACM Reference Format:
Weihao Cui*, Han Zhao*, Quan Chen*◇, Ningxin Zheng†, Jingwen Leng*◇, Jieru Zhao*◇, Zhuo Song‡, Tao Ma‡, Yong Yang‡, Chao Li*◇, Minyi Guo*◇. 2021. Enable Simultaneous DNN Services Based on Deterministic Operator Overlap and Precise Latency Prediction. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458817.3476143

1 INTRODUCTION

Deep Neural Network (DNN) empowers intelligent latency-critical (LC) services (e.g., computational vision [19, 39, 46, 47], natural language processing [16, 20], recommendation system [49]). While DNN-based latency-critical services are computationally demanding and have stringent Quality-of-Service (QoS) requirements, the common practice is deploying a single service on a dedicated GPU and routing the loads to each GPU using a cluster-level load balancer. A large number of dedicated DNN serving systems [12, 15, 17, 40, 44] (e.g., Triton [29], Clipper [13], TF-Serving [32]) have been proposed to run DNN inference queries on GPUs.

However, when the load of an LC service is low (services often experience diurnal load patterns [8]), GPUs suffer from low utilization. Prior works like Prema [11], Clockwork [17], Nexus [44], and PipeSwitch [4] have shown that running multiple applications simultaneously on a GPU is capable of improving the utilization. Meanwhile, NVIDIA introduces MPS (Multi-Process Service) [26] and MIG (Multi-Instance GPU) [25] for enabling concurrent sharing of a single GPU among multiple applications.

Obviously, running multiple services simultaneously on a single GPU increases the end-to-end latencies of user queries[1]. This longer latency is acceptable if the latency increase is stable and shorter than the QoS target. **However, simply running multiple DNN services simultaneously results in unstable long latency, risking the QoS violation of the co-located services.** We find that the unstable long latencies originate from two main reasons. **1) The latency of a query is sensitive to the input.** The queries may be executed in different batch sizes and have different sequence lengths. **2) The operator/kernel overlap is not deterministic.** The user queries are submitted and received at a random time. How the simultaneous user queries overlap with each other is not known before they are actually processed. When the operators of simultaneous queries show different contention degrees, the latency increase is not known.

Some prior works enable simultaneous services on both CPU servers [9, 36] and GPU servers [17, 44, 53]. For works on CPUs, computational resources are reallocated to the services dynamically. They rely on low overhead resource reallocation techniques (e.g., core affinity, cache allocation, memory bandwidth allocation) and are not applicable for co-locating services on GPUs due to the large overhead. GPUs are non-preemptive, and the overhead of reallocating resources is heavy.

Nexus [44] and Clockwork [17] are the most related works that deploy multiple services on a GPU. To resolve the nondeterministic operator overlap, the runtime serving system on a GPU **sequentially** processes the operators of the user queries. The operators are

Quan Chen and Minyi Guo are corresponding authors.

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476143

[1] A query is the processing of a user request.

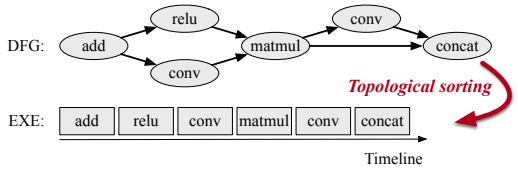

Figure 1: Processing a query of a DNN service.

executed in First-Come-First-Serve (FCFS), Shortest-Job-First (SJF), or Earliest Deadline First (EDF) manner. In this case, the operators are actually not overlapped at all, and the response latencies of user queries are deterministic and predictable. However, the sequential co-location manner results in low throughput, as the GPUs are not fully utilized with a single operator [27, 53].

There is an opportunity to further improve the processing speed for the simultaneous DNN services if the operators run simultaneously and the latency increase can be precisely predicted. We face three challenges in taking advantage of the opportunity. **1) How to make the operator overlap deterministic?** A mechanism is required to configure the operator co-location at runtime instead of allowing the operators to overlap randomly. In this way, we are able to control the response latencies of the queries for QoS. **2) How to predict the latency increase due to the overlap?** The operators that run simultaneously contend for shared resources (e.g., global memory bandwidth on GPU), but they have different sensitivities to the contention. **3) How to guarantee the QoS of all the simultaneous DNN services?** All the user queries of the simultaneous DNN services have QoS requirements. A low overhead online scheduling policy is required to carefully manage the operators of all the queries to ensure the QoS of all the services.

To resolve the above challenges, we analyze the way that a query is processed. Figure 1 shows an example DNN model (expressed to be a data flow graph, DFG) and the way it is processed. A DNN query is processed by executing a series of operators (e.g., relu, conv) sequentially in the topological order instead of invoking a single huge function. In this case, By dividing the processing of a query into several parts and controlling the time of issuing the parts of each query, the operator overlap is determined.

Besides, our investigation shows that the latencies of a query in multiple runs are stable if its co-located operators are determined. The detailed experimental setup is described in Section 5.2. In our 40,000 runs of 21 co-location pairs, the average standard deviation of the queries' latencies is 0.65ms, while the 90%-ile of the standard deviation is 1.58ms. The deviation is small compared with the queries' average latency (4.53%). Therefore, the latency of a DNN query is predictable if its input is known and the operator overlap is determined. There is a chance to predict the co-running latency by determining the operator overlap.

Based on the above two insights, we propose **Abacus**, a runtime system armed with precise latency prediction to enable simultaneous DNN services on GPUs. Abacus consolidates the determinism by issuing the deterministic operator group per round of scheduling. Abacus composes of an *overlap-aware latency predictor*, a *headroom-based query controller*, and a *segmental model executor*. The *latency predictor* is built using an MLP (Multilayer Perceptron) model. The predictor can precisely predict the latency given a fixed operator schedule group from already known DNN services. The *query controller* is responsible for guaranteeing the QoS of all the simultaneous DNN services. For each round of scheduling, The *query controller* schedules a group of overlapped operators based on each queries' headroom to the QoS target. The headroom-based scheduling does not provision the GPU for the query with the least headroom. Instead, it tries to form a DNN operator group from all queries for scheduling. Hence, the controller searches for the optimal operator group to issue under the constraint of the least headroom of all queries. A multi-way searching is conducted to add as many operators as possible to the group while guaranteeing the QoS of the query with the least headroom by consulting the *latency predictor*. *A segmental model executor* processes the queries with the configuration of the optimal operator group. The completely processed queries return the results, and the partially processed queries participate in the next round of scheduling.

The main contributions of Abacus are as follows:

- **Comprehensive analysis of the unpredictability in simultaneously deployed DNN services.** We identify the two leading root causes for the unpredictability with simultaneous DNN service deployment enabled. The analysis motivates Abacus.
- **Design of a precise latency prediction model for operator groups.** We build an accurate model to predict the latency of an operator group. The operators in an operator group are from multiple DNN services and run simultaneously.
- **Design of a deterministic operator overlap mechanism.** The mechanism does not bring in extra overhead like synchronization for the determinism on non-preemptive GPUs.

Our evaluation using seven popular DNNs on an Nvidia A100 GPU shows that Abacus reduces 51.3% of the QoS violations for simultaneous DNN services and improves the throughput by 29.8% on average compared with state-of-the-art solutions.

2 RELATED WORK

Addressing the QoS problem for simultaneous DNN services on a single GPU is challenging. Nexus [44] and ClockWork [17] enable simultaneous DNN services deployment through cluster-level management. However, from the perspective of a single GPU, Nexus and ClockWork provision the GPU for a single DNN service at a certain moment without improving throughput by DNN operator overlap. In evaluation, we compare Abacus with the default scheduling policy (FCFS, SJF, EDF) used in Nexus and ClockWork.

Baymax [8], Prophet [7], and Laius [52, 53] address the QoS problem for co-locating latency-critic(LC) jobs and best-effort(BE) jobs on GPU. These works perform per-kernel scheduling and sacrifice the performance of BE jobs for accelerating the LC jobs. In our scenario, per-kernel scheduling incurs high overhead, and we need to handle the QoS problem given multiple services.

There are some other works done on simulators. Works like Themis [50, 58, 59], and HSM [60] model the slowdown of co-locating applications on GPUs. The predictors used in these works need the underlying hardware information for predicting the interference of multiple services, which is impossible for online scheduling in production. Some works are related to processing multiple

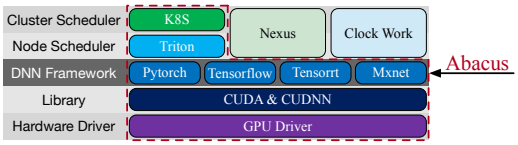

Figure 2: Layered software stack of DNN Serving. Abacus enables simultaneous DNN services without modifying the upstream cluster-level management.

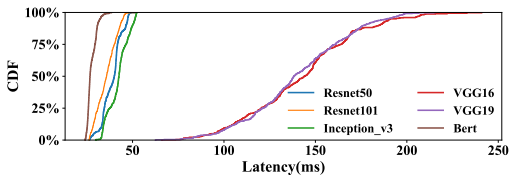

Figure 3: The latency distribution of the queries of *Resnet152* when it co-runs with other DNN services using Nvidia MPS.

DNNs on the same accelerators [3, 11]. Prema [11] addresses the simultaneous deployment of DNN services on a systolic array simulator [10, 41]. The foundation of Prema is the preemptive mechanism of the simulator. However, GPUs in production are non-preemptive. Moreover, Prema provides no strict QoS guarantee for queries.

On top of the above works on a single GPU, some works focus on the orchestration of DNN serving at the cluster level [37, 48, 51, 57]. Mark [51], and Swift [37] provide cost-efficient and QoS-aware orchestration for single DNN services. Co-location is not involved in improving throughput. Kube-knots [48] orchestrates the GPU containers at the cluster level for enabling LC jobs and BE jobs. It predicts the inter and intra-application correlation for QoS guarantee before co-locating the relevant containers. It is not able to guarantee the QoS of each LC query while co-locating LC jobs.

For co-location of multiple services on traditional CPU machines, several works like Parties [9], CLITE [36], Sinan [55], Avalon [6], Ursa [54], Sturgeon [33, 34], Bats [56] have been proposed to achieve high throughput and hardware utilization for traditional services [2, 21]. Techniques used in them are not applicable in GPU-based DNN services. Basically, CPU-related resources can be reallocated without interrupting the running services [1]. On GPUs, termination of the running process is essential for reconfiguring the allocated resources for DNN services.

Some works like Ebird [14, 15], and C.Guo *et al.* [18] exploit multiple CUDA streams [24] for improving hardware utilization of the single DNN service. However, they are not capable of guaranteeing the QoS while deploying multiple different DNN services simultaneously. Some compiler-level works (e.g., Rammer [23], TensorRT [31]) enable intra-operator parallelism for accelerating the DNN inference. In general, these compilers fuse multiple operators into a single GPU function for providing stable high performance. These works are not the opposite of the way that Abacus processes the DNN query.

3 BACKGROUND AND MOTIVATION

In this section, we show the long tail latency problem of running multiple DNN services simultaneously, and discuss the root causes of this problem.

3.1 Philosophy of Abacus

Figure 2 shows the general software stack of DNN serving. Prior researches like Nexus [44] and Clockwork [17] schedule multiple DNN services on a cluster by exploiting cluster-level query routing. On the contrary, we design and implement Abacus to manage queries in the lower framework level.

For the works at the cluster level [17, 44], a central controller is required to route queries of different services to different GPUs inside the cluster. Network bandwidth becomes the bottleneck [17], and they are not portable with popular DNN service schedulers like Kubernetes [22] and Triton [29]. In addition, although multiple DNN services are deployed on a GPU cluster, the cluster-level works use one of FCFS, SJF, and EDF policies to run the queries sequentially on each GPU for ensuring predictive QoS guarantee [17, 44]. The sequential execution manner results in unnecessary long queuing time, as the GPUs are not fully utilized with a single operator [53]. They support low peak throughput with QoS constraint due to the poor GPU utilization (proved by the experiments in Section 7.2).

To overcome the above weakness, Abacus exploits the runtime operator scheduling of DNN frameworks instead of modifying the cluster-level management. While the schedulers guarantee that the load does not exceed the hardware capacity, Abacus guarantees the QoS of each individual query with simultaneous DNN services enabled. **As Abacus does not rely on upstream cluster managers, it can be used with existing cluster-level schedulers, like Kubernetes, for large-scale in-production deployment.** Experiments in Section 7.6 show that by integrating Kubernetes with Abacus, DNN services achieve much higher throughput compared with state-of-the-art cluster-level DNN scheduler Clockwork [17].

3.2 Long Tail Latency

Without cluster-level throttling [17, 44], multiple queries from the co-located DNN services may arrive at the GPU simultaneously. In this subsection, we exploit whether the MPS technique of Nvidia can effectively host simultaneous DNN services on a single GPU.

In this experiment, we run a DNN service *Resnet152* with another DNN service on an Nvidia A100 GPU and report the latency distribution of its queries in Figure 3. In the experiment, We fix the batch size of Resnet152 to be 32 and the image size to be 224×224 to create a stable load. The experiment is conducted in a close-loop way, excluding the queuing time in a real serving system. In this case, the end-to-end latencies of the *Resnet152* queries are stable in solo-run mode. The inputs of its co-located services change dynamically, and the query arrival rate follows the Poisson distribution. In the figure, the line "VGG16" represents the latency distribution of *Resnet152* when it runs with *VGG16* simultaneously. The software and hardware setup is presented in Section 7. Experiments with other services show similar results.

As shown in Figure 3, the latencies of *Resnet152* queries vary significantly when it runs with other DNN services simultaneously. The latencies range from 24 milliseconds to more than 241 milliseconds, while the solo-run latency is 24 milliseconds. The latency of

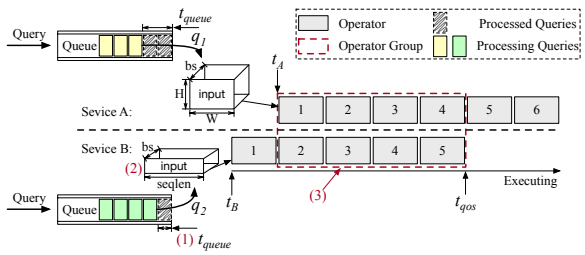

Figure 4: Unstable latencies of the queries when two or more DNN services run simultaneously.

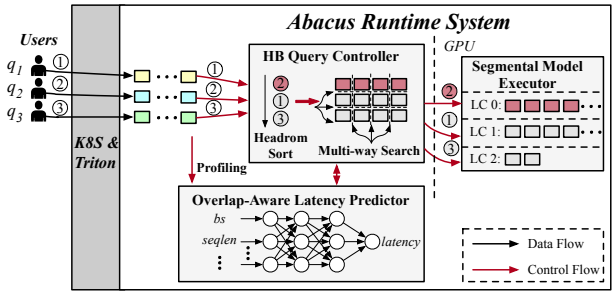

Figure 5: Design architecture of Abacus.

a query is impacted by its own load, its co-runners, the co-runners' load, and how the queries overlap. Compared with other co-runners, VGG16 and VGG19 result in much longer latencies.

For a DNN service that interacts with end-users, it is critical to make sure that its tail latency (e.g., 99%-ile latency) to be below the pre-defined QoS target (e.g., 50ms). *It is not safe to directly run multiple DNN services on a GPU with MPS, as we cannot predict the latency of a query before it completes.*

3.3 Unstable Latencies of Queries

The longer latency of a query when it runs simultaneously with other queries is not a problem if it is shorter than the QoS target. As shown in Figure 3, the latencies of some queries are much shorter than the QoS target, while other queries (some queries of VGG16 and VGG19) suffer from QoS violation. The unstable query latencies disable the opportunity to run multiple DNN services simultaneously on a GPU.

We, therefore, analyze the reasons for the unstable latencies. **The root reasons are: 1) the queries arrive in irregular time intervals; 2) the operators contend for shared resources.** In this case, the queries have different overlaps with other queries and suffer from different performance interference.

Figure 4 illustrates the way that two services (Service-A and Service-B) run on a GPU simultaneously. In the figure, Service-A uses a computer vision (CV) model, and Service-B uses a natural language processing (NLP) model. As observed, the end-to-end latencies of Service-B's queries are not identical because:

(1) **Queries have different queuing time.** The query may queue up for the hardware resources or contend for PCI-E, NVLink to transfer data. The queuing time and data transfer time are affected by loads of the simultaneous DNN services. When the load is high, the queuing and data transfer time is longer.
(2) **Queries have different inputs and batch sizes.** The input size determines the processing time of a query, and DNN services often accept queries with inputs of different sizes. For instance, the input image can be of different resolutions for Service-A, and the sequence length also varies for Service-B. Besides, the batch size of query processing is determined by the upper-level load balancer at runtime. Both different inputs and batch sizes result in different processing time.
(3) **Concurrent queries have non-deterministic overlap.** The operators 1-5 of the current query of Service-A are overlapped with the operators 2-6 of the current query of Service-B in Figure 4. The overlap increases the response latencies, as the overlapped operators share and contend for the resources in the GPU. However, previous and later queries may have totally different overlaps, because queries arrive in irregular time intervals. The non-deterministic overlap results in the different latency increases of the queries.

It is profitable to delay several operators of q_1 to reduce the operator overlap in Figure 4, if the latency of q_1 is much shorter than its QoS target and q_2 suffers from QoS violation. For instance, if we can delay operators 4-5 of q_1 until q_2 completes, the latency of q_2 is reduced due to shorter overlap. Meanwhile, the slight latency increase of q_1 does not lead to its QoS violation. **It is challenging to determine the overlapped operators so that both $q1$ and $q2$ can satisfy the QoS targets.**

To achieve the above purpose, we have to precisely predict the latencies of the queries with different input sizes and overlap situations. However, for q_2, it knows neither the inputs nor the start time of q_1 with the current MPS-based co-location solutions. The end-to-end latencies of queries in the simultaneous-DNN-service scenario are currently not predictable. A mechanism is required to obtain all those runtime information, predict the latencies with different overlap options, and stabilize the appropriate overlaps for each query at runtime.

4 DESIGN OF ABACUS

Since the factors that affect the query's latency for simultaneous DNN services are only known at runtime, we propose **Abacus** to be a runtime operator management system at the framework level.

Figure 5 presents the overview of Abacus. It is comprised of an *overlap-aware latency predictor*, a *headroom-based query controller*, and *segmental model executors*. The latency predictor precisely predicts the processing time of an *operator schedule group* in which the operators are issued to run on the GPU in parallel. The query controller determines the processing order of the received queries, and identifies the optimal operator schedule group for each query. A segmental model executor processes a query's corresponding operators in an operator schedule group. In more detail, Abacus runs multiple DNN services simultaneously in the following steps.

(1) Abacus sorts the received queries based on the latency headrooms to their QoS targets in ascending order. Let q_2 in Figure 5 represent the query that has the shortest headroom to its QoS target. Abacus schedules the operators to ensure its QoS first.

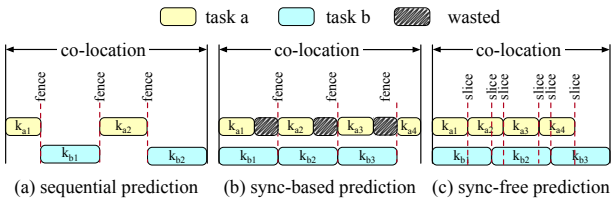

Figure 6: Three kernel-level duration prediction methods.

(2) If q_2 still has latency headroom to the QoS target, Abacus identifies the optimal operator group that can run with q_2 without incurring QoS violation. For an operator group candidate, Abacus predicts the latency of q_2 if the operator group runs with q_2 simultaneously. The group that conveys the largest throughput without incurring q_2's QoS violation is the optimal one. Abacus uses multi-way search to make the prediction for multiple operator group candidates (Section 6.3). The prediction is made based on a precise offline trained duration model (Section 5.5).
(3) Once the operator schedule group is identified, the query controller launches it only if the segmental model executor is idle for ensuring the deterministic of the former scheduled operator group. The segmental model executor then processes the queries' corresponding operators in the group. The segmental model executor saves the intermediate results for those partially processed queries.
(4) Once q_2 returns, Abacus identifies the next query with the shortest latency headroom to its QoS target, and searches the optimal operator group for it.

Several challenges have to be resolved in Abacus. Only when the predictor is accurate and fast, the query controller is able to identify the optimal operator schedule group in time. The query controller also needs to minimize the number of tries needed to find the optimal group, as each prediction takes time. The segmental model execution has to parse and resume the execution of a query with low overhead. Otherwise, the overhead itself may already result in the QoS violation.

Given N DNN services, Abacus only trains a single model for the latency predictor to predict the duration of an *operator schedule group*, no matter which of the N services run simultaneously. Our experiments show that it is not necessary to train separate models for different co-location pairs as prior works do [8, 53].

5 OVERLAP-AWARE LATENCY PREDICTION

In this section, we first discuss the reason for performing the prediction in operator group granularity. Then, we show that it is possible to predict an operator group's latency, although the operators may be overlapped. Lastly, we detail the way to select features, collect training samples, and identify appropriate prediction techniques.

5.1 Incapable of Prior Kernel-level Prediction

In order to guarantee the QoS of a query, we have to predict its latency if it co-runs with other services. A straightforward way is predicting the duration of each kernel in the query and aggregating the durations to be the query's duration. Prior researches like Prema [11] adopt this way.

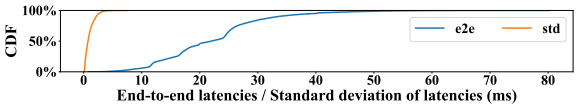

Figure 7: Statistics of collected training samples.

Figure 6 shows three kernel-level duration prediction methods. As shown in Figure 6(a), Prema [11] inserts a synchronization fence between kernels and runs the kernels sequentially. In this way, the duration of a query can be predicted by aggregating the solo-run time of the kernels. This method disables the kernel overlap, thus results in low throughput.

An improved idea is enabling the overlap, making the kernel overlap deterministic by adding explicit fences and predict the duration of the overlapped kernel pairs, as shown in Figure 6(b). However, the overlapped kernels have different processing time, and the throughput is still not maximized.

Without hurting the throughput, an optimal way is to predict the duration when allowing the kernels to overlap freely, as shown in Figure6(c). For instance, some prior simulator-based researches, such as HSM [60] and Themis [59], predict the slowdown of each overlap slice based on performance event counters. However, they are not applicable in real systems as such events are not available at runtime in the in-production GPUs. In addition, they are posteriori methods and are not able to predict the kernel duration proactively. Proactive prediction is necessary to determine whether two kernels should be overlapped.

Our measurement also shows that the time needed to perform the above duration prediction takes $0.1ms$ in real systems, while the duration of kernels/operators in DNN models often have the same order of magnitude. Kernel-level duration prediction introduces too heavy overhead to be applied in real systems.

Therefore, we introduce *operator group* for flexible kernel overlap, high prediction accuracy, and low prediction overhead in general. Specifically, Abacus groups the overlapped DNN operators into operator groups and predicts the duration of each operator group. Abacus flexibly determines the DNN operators that can be safely overlapped proactively.

5.2 Latency Determinism of Operator Group

The operators from multiple models in an operator group may have different overlaps in different runs. It is crucial to find whether the latency of an operator group is deterministic or not.

To this end, we generate 42,000 operator groups by co-running 7 DNN models in a pair-wise manner. The detailed method to generate the operator groups is shown in Section 5.4. For the 42,000 operator groups, we run each group 100 times and collect the corresponding latencies. Figure 7 reports the latency distribution of the operator groups, and the standard deviations of the operator group latencies at different runs.

As observed, the standard deviations are shorter than $1ms$, while the actual latencies of the operator groups are much longer. The average latency (T_{e2e_avg}) is $15.9ms$, while the 90%-ile of the latency is $25.8ms$. For operator groups generated from triplet-wise

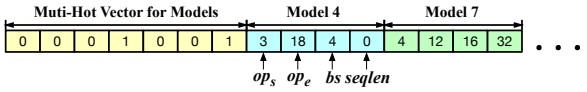

Figure 8: Input features for training a duration model for operator groups of pair-wise model co-running.

co-location and quadruplet-wise co-location show similar results. The latency of an operator group is deterministic and predictable.

5.3 Determining Representative Features

Optimally, we can use the runtime performance events (e.g., cache misses, global memory bandwidth) [60] and the inherent features of the operators as the inputs to build a duration model for operator groups. However, Nvidia does not provide an interface to obtain runtime events online. The hardware events reported by tools like *nvprof* [30], *nsight compute* [28] can only be obtained after the application completes. It is not applicable to rely on runtime hardware events to build the model for in-production GPUs, as such events cannot be obtained during the online scheduling.

Some readers may think that we can build a performance model for each individual operator, and calculates the duration of an operator group accordingly. However, the operators in an operator group may run either sequentially or in an overlapped manner, depending on the resource requirement of each operator, as well as the issuing order of the operators [7]. Building a model for each operator is not able to capture such unstable overlap behaviors.

We, therefore, collect the durations of operator groups and the corresponding representative features to train such duration model. Suppose there are N possible models that may co-run on a GPU. For each model, its operators are numbered with their topological order in the model. There are many operators in DNN models (e.g., 241 operators for Resnet101). A large input feature will be generated if Abacus selects the information of all operators as the input feature like previous work [8, 11]. Figure 8 shows an example input feature in Abacus for training a duration model for operator groups when two of the N models co-run. In the input feature vector, a N-bit bitmap is used to indicate which two models co-run. Besides, for each of the co-running models, the feature vector also indicates the start operator op_s, the end operator op_e, the batch size bs, and the sequence length $seqlen$ in the operator group sample.

Note that the design of the bitmap unifies the information about the overlapped operators inside an operator group, and $seqlen$ is only used for Bert-like DNN models[2].

5.4 Improving Sampling Efficiency

We use 6 CV models and 1 NLP model to collect the training samples. The queries' batch size is randomly chosen to simulate the input non-determinism for all models. For Bert, the sequence length is also involved in randomization. Table 1 shows the details of the models used to collect the samples.

The sample space increases exponentially as the involved models increase. Naive sampling results in a huge sample space that results in too long offline time. We analyze the creation of operator groups in Abacus. There are two principles when Abacus builds operator groups. First of all, in an operator group, at least a query completes. Second, a new query may be received, and its operators may be added to the current operator group.

[2]The sequence length used here is only for Bert-like DNN models. Traditional NLP models like LSTM [20] do not need the sequence length information, which is related to the number of operator and included in the first two elements of model vector

Table 1: DNN models used for serving.

Models	Random Parameters
Resnet50 [19]; Resnet101; Resnet152; Inception_V3 [47]; VGG16 [46]; VGG19	Batch Size: [4, 8, 16, 32]
Bert [16]	Batch Size: [4, 8, 16, 32] Seq Length: [8, 16, 32, 64]

Figure 9: Generating the operator group through instance-based sampling.

Hence, we adopt instance-based sampling [45] to improve the sampling efficiency. More specifically, we sample the operator groups in the same principles. Figure 9 shows the way we generate the operator group samples. (1) We first randomly choose the number of models to complete in the operator group. In Figure 9, 2 models (Model A, Model B) are selected. (2) Then, we randomly choose the number of the newly arrived models that are processed from the first operator. In Figure 9, 2 models (Model B, Model C) are selected. (3) After the above two steps, the operators of Model B for the operator group are determined. The beginning operator of Model A and the end operator of Model C are randomized.

In this way, we get samples that exist in the real scheduling of Abacus. The sampling efficiency and the predictor's accuracy are improved. In addition, the same number of operator groups are sampled for each possible batch size. Our statistics show that Abacus takes 42 hours to collect $2000 \times C_7^2 = 42,000$ samples (each sample run 100 times) for pair-wise service co-location.

5.5 Determining Modeling Techniques

We evaluate three widely-used prediction methods, Linear Regression (LR) [42], Support Vector Machines (SVM) [5], and Multilayer Perceptron (MLP), in training the duration model. We limit the hidden layer of the MLP model to 3 layers, whose dimension is 32.

We randomly choose 80% of the collected 40,000 samples in Section 5.4 to train the model and using the rest for testing. As mentioned in Section 5.4, we have 21 pair-wise co-location combinations. There are two possible options: 1) train a duration model for each co-location combination; 2) train a unified duration model for all the combinations. The latter option consumes fewer resources but may result in low accuracy.

Figure 10 shows the prediction errors of both options. In the figure, the column "all" shows the prediction errors of the unified duration models. Here, the prediction error is measured by the

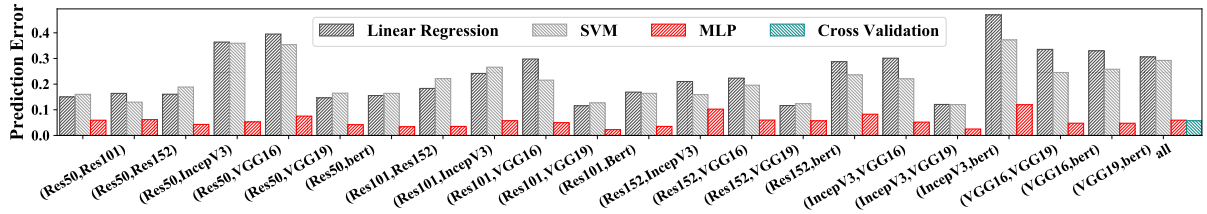

Figure 10: Prediction errors of all the evaluated modeling techniques: Linear Regression, SVM, and MLP. We also show the cross validation accuracy of MLP.

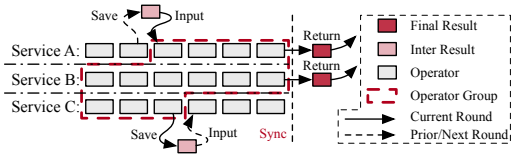

Figure 11: Executing queries in a deterministic way with the flexible segmental model executor.

mean absolute percentage error, as shown in Equation 1.

$$Error = \frac{|Predicted\ Latency - Real\ Latency|}{Real\ Latency} \quad (1)$$

Specifically, if a model is trained for each combination, LR and SVM achieve prediction errors of 23.5% and 21.5% on average. For VGG-related pair-wise co-location, the errors of LR and SVM are up to 47.0% and 37.2%. For all combinations, the MLP model shows consistently high accuracy (the average error is 5.5%).

If a unified model is trained for all combinations, the errors of LR and SVM are 30.1% and 29.2%, and the error of MLP is 5.7%. The cross-validation shows that MLP always has high accuracy in predicting the duration of operator groups. The bar "Cross Validation" in Figure 10 shows the prediction error of the cross-validation. Besides, for triplet-wise co-location and quadruplet-wise co-location, the prediction errors of the unified duration model trained predictor are 4.9% and 6.4%, respectively.

6 HEADROOM-BASED SCHEDULING

In this section, we first introduce the flexible segmental model executor. Then, we show the way to perform headroom-based scheduling based on the latency predictor.

6.1 Flexible Segmental Model Executor

Abacus controls the query execution of simultaneous services by issuing operator groups. The whole execution of a query can be divided into several segments. The query controller requires a flexible model executor to perform the headroom-based scheduling. Hence, Abacus designs the segmental model executor to satisfy the requirements for executing the operator schedule group.

Figure 11 illustrates the segmental model executor. First of all, each DNN service is deployed in a separate process inside the model executor for the purpose of privacy and avoiding the chain reaction of the crash. The segmental model executor controls the execution of the operator group through communicating with each DNN service. After receiving an operator schedule group, the model executor notifies corresponding processes for executing operators included in the operator group. Then the model executor waits for all processes' completion and replies to the query controller. The model executor works in an exclusive way to guarantee the **determinism** of executed operator group.

During the processing of the operator group, there are a few things to deal with. As depicted in Figure 11, only the first three operators are executed in the current round of scheduling for queries like Service C. The model executor saves the intermediate results for these queries. For queries like Service A, the first operator has been processed in the former round of scheduling. Then, the model executor restores the input from early saved intermediate results. A synchronization operation on GPU is needed before replying to the query controller for determinism, and the final results of those fully processed queries are returned.

6.2 Headroom-based QoS Guaranteeing

Abacus has to guarantee the QoS of DNN services deployed simultaneously on a GPU. Abacus achieves the goal of QoS guarantee under the scheduling of query controller. Instead of directly considering the QoS of all DNN services, the query controller only guarantees the QoS of one query for each round of scheduling. Like the Earliest Deadline First scheduling (EDF), the query controller prioritizes the query with the earliest deadline. However, the query controller forms an operator group based on the deadline priority rather than schedule the query with the highest priority.

$$\begin{aligned} T_h &= T_{QoS} - T_{queue} - T_{comms} - T_{completed} \\ &= T_{QoS} - (T_{cur} - T_{start}) \end{aligned} \quad (2)$$

The query controller first calculates the QoS headroom (denoted by T_h) in Equation 2. The headroom is obtained by subtracting the queueing time (T_{queue}), the data transfer time through PCI-E and/or NVLink (T_{comms}), and the duration of completed operators ($T_{completed}$) from the QoS target (T_{QoS}). In the equation, we already know the start timestamp (T_{start}) of each query and the current timestamp (T_{cur}). Hence, in the second row of the equation, $T_{cur} - T_{start}$ already contains T_{queue}, T_{comms}, and $T_{completed}$.

Then the controller sorts all queries according to the QoS headroom in ascending order and then searches for the operator group under the constraint of the least QoS headroom. In the current round of scheduling, the query controller ensures the execution of the query with the least QoS headroom by adding all its operators to the candidate operator group. The query controller tries to add

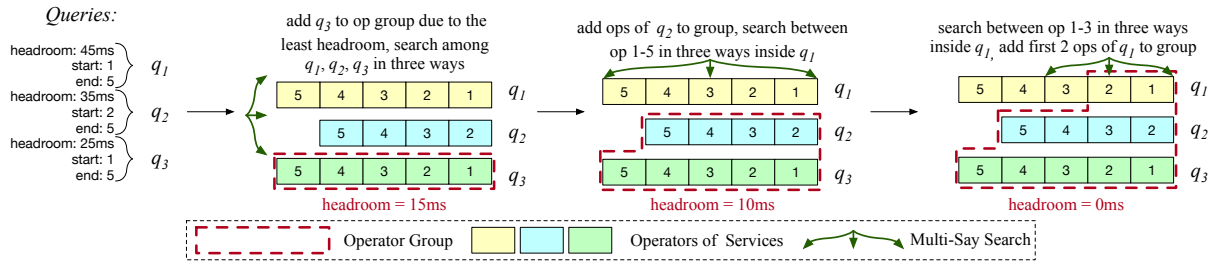

Figure 12: Generating an operator schedule group based on the QoS headrooms of all the active queries.

as many operators as possible to the operator group as long as the predicted latency given by the overlap-aware latency predictor does not exceed the QoS headroom. The added operators from the rest queries are selected in the order sorted by their QoS headrooms.

Figure 12 illustrates an example of scheduling three queries by the query controller. The headrooms of the queries waiting for scheduling are (q_1:45ms; q_2:35ms; q_3:25ms). Therefore, the headroom (25ms) of q_3 is the headroom for QoS guaranteeing in this round of scheduling. Then the controller directly adds the operators of q_3 into the operator group, and leaves the QoS headroom of 15ms, predicted by the latency predictor. Because q_2's headroom is smaller than that of q_1, the controller first considers adding operators of q_2 to the operator group, and only 10ms of headroom is left for q_1. After adding the first two operators of q_1, none of the QoS headroom is left. Finally, the operator schedule group is generated and ready for execution. The operators of q_1 that did not enter the operator group will be scheduled in the next round of headroom-based scheduling.

The query controller also adopts a drop mechanism to avoid the QoS violation. If the latency predictor predicts that the headroom is not enough for finishing the execution of the query with QoS target for this round of scheduling, the query controller directly drops the query and enters the next round of scheduling. Keeping processing causes the QoS violation of current and later queries.

6.3 Identifying the Optimal Operator Group

Identifying the optimal operator group requires scanning operators and predict the duration of different possible operator groups. Although making a prediction only takes 0.06ms, it is still time-consuming to try all the possibilities. To this end, we adopt multi-way search to speed up the searching, and pipeline the operator scheduling and operator execution to hide the scheduling overhead.

Multi-way search. The process of one headroom-based scheduling requires many times prediction. Figure 12 shows an example of adding 6 DNN operators into the operator group for q_2 and q_1. In this example, the latency prediction is done 6 times sequentially, resulting in a slow search. We notice that each latency prediction is independent. Based on this observation, we conduct a multi-way search for accelerating the exploration of the optimal operator group by feeding the duration model with batched input features for computation all at once. The multi-way search consumes more computation, as the batched computation of MLP may need more CPU cores. Our experimental data in Section 7.7 shows that a single core is enough for the multi-way search.

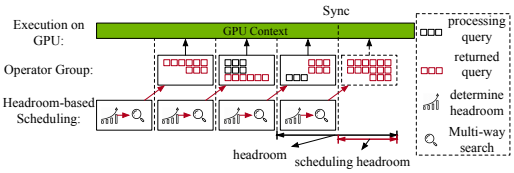

Figure 13: Pipelined headroom-based operator scheduling.

More specifically, for the three queries q_1, q_2, and q_3 in Figure 12, we first search between queries in three ways instead of trying to add operators directly. After the first prediction, we know that the operators of q_2 and q_3 can be added to the operator group. The query controller continues to conduct the search in three ways among the operators of q_1. After two predictions, the query controller completes the exploration of the optimal operator group.

Pipeline scheduling and execution. Execution on GPU is asynchronous with the operation of the host side. We leverage this property for generating an operator group while executing the former operator group. Figure 13 shows the detailed scheduling process. After issuing the operator group, the query controller immediately starts the next round of headroom-based scheduling. Since the latency of the scheduled operator group is often larger than that of the searching process, the overhead is hidden.

Note that we need to update the headroom used for searching the operator group. The headroom ($T_{schedule\_h}$) used for forming the new operator group is calculated by Equation 3.

$$T_{schedule\_h} = T_h - T_{predict\_lat} \tag{3}$$

Because the identified operator group is not issued until the GPU completes the former one, the predicted latency of the former operator group ($T_{predict\_lat}$) needs to be subtracted from the QoS headroom (T_h) for scheduling. Otherwise, the query would not return before the required QoS target.

It is worth noting that Abacus does not increase the total times of synchronizations during the pipelined scheduling. As observed in Figure 13, at least a query (selected for QoS guarantee by query scheduler) returns for each time of synchronization in each round of Abacus's scheduling. Due to the asynchronization feature of GPUs, each query in sequential scheduling also needs an explicit synchronization for confirming the completion of execution. Therefore, a DNN query may be divided into several operator groups, but the total synchronizations do not increase. The pipeline scheduling does not introduce extra synchronization overhead.

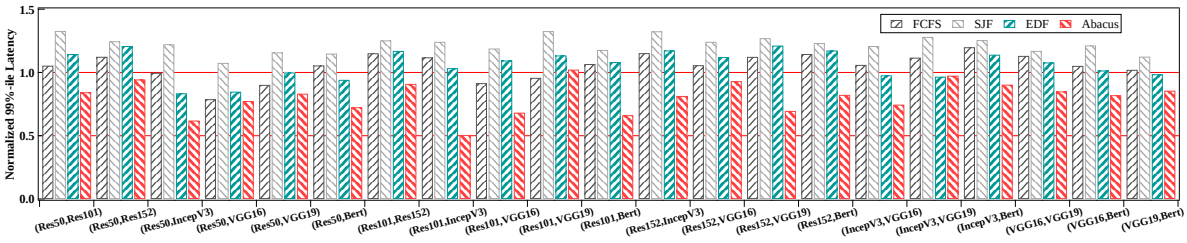

Figure 14: End-to-end 99%-ile latency of each pair-wise co-location with FCFS, SJF, EDF, and Abacus.

Table 2: Evaluation specifications.

Hardware	CPU: Intel(R) Xeon(R) Silver 4210R (2.4GHz); GPU: Nvidia A100 (128 SMs)
Software	OS: Ubuntu 20.04.1 (kernel 5.8.0); GPU Driver Version: 460.39; CUDA Version: 11.2, CUDNN Version: 8.1

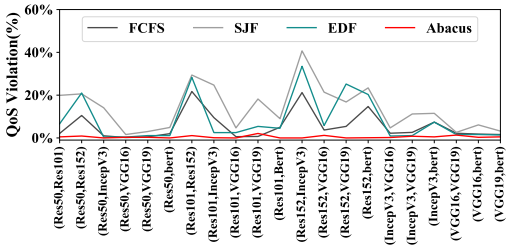

Figure 15: QoS violation ratio of each pair-wise co-location with FCFS, SJF, EDF, and Abacus.

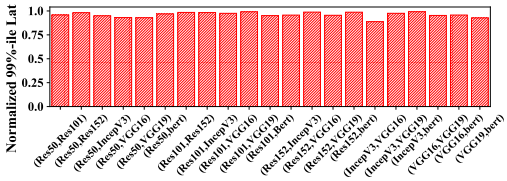

Figure 16: 99%-ile latencies with small co-located DNNs

7 EVALUATION

In this section, we evaluate the effectiveness of Abacus in ensuring QoS and improving throughput. Pair-wise deployment is firstly measured in detail. We also extend Abacus to more services and integrate it with MIGs. The techniques like multi-way search and pipelining scheduling and execution are also evaluated. In the end, we discuss the overhead of Abacus.

7.1 Experiment Setup

We have implemented Abacus based on Pytorch [35]. All the evaluations are performed on a machine that equips the latest Nvidia Ampere 100 (A100) GPU. On the GPU, MPS is enabled for sharing GPU resources between the deployed services. More details about the experiment environment are shown in Table 2.

Table 1 shows the widely-used 7 DNN inference models from image processing and natural language processing fields used to evaluate Abacus. For easing of description, we use (A, B) to represent the case that A and B run on a GPU simultaneously. $(A, B) + (C, D)$ means that four services are divided into two deploy groups. The two groups are deployed on a single GPU using MIG [25] that divides an A100 GPU into multiple smaller instances.

We compare Abacus with FCFS (First Come First Serve), SJF (Shortest Job First), and EDF (Earliest Deadline First), the scheduling policies used in Clockwork [17], and Nexus [44] for co-locating multiple services. Besides, for a fair comparison, we enable the query-drop mechanism [44] for the baselines to improve their QoS. After adopting the mechanism, a queued query is directly dropped if its latency is already over the QoS target and is not counted in the latency experiment for the baselines. The batch size and sequence length of each query are randomly selected in Table 1. The service load is generated using MLperf [38], and the arrival time pattern satisfies the Poisson distribution. Same as prior DNN inference work [7, 44], the QoS targets of the DNN services are set to be 2× their solo-run latencies with the maximum inputs (ranging from 50 to 150 milliseconds).

7.2 Ensuring QoS

We first evaluate the effectiveness of Abacus in ensuring the QoS of the pair-wise co-located services. In this experiment, the queries are submitted at a load of 50 queries-per-second (QPS). This load does not saturate the GPU and better reveals whether Abacus can ensure the QoS and reduce tail latency.

Figure 14 presents the 99%-ile latency of all the $C_7^2 = 21$ DNN service co-location pairs normalized to the QoS targets. Observed from this figure, Abacus reduces the 99%-ile latency by 23.1%, 34.1%, 23.8% on average compared with FCFS, SJF, and EDF, respectively. If the query-drop mechanism is disabled, FCFS, SJF, and EDF result in much longer tail latency.

Moreover, Figure 15 shows the percentage of the queries that suffer from QoS violation. Abacus reduces the percentages of the queries that suffer from QoS violation by 38.8%, 71.0%, 44.0% on average compared with FCFS, SJF, and EDF. There is almost no query that suffers from QoS violation with Abacus. In this figure, the dropped queries are counted to reveal the real user experience.

As shown in Figure 15, FCFS, SJF, and EDF result in the most QoS violations for (Res152, IncepV3). This is because most operators of Res152 and IncepV3 are convolution operators that have small

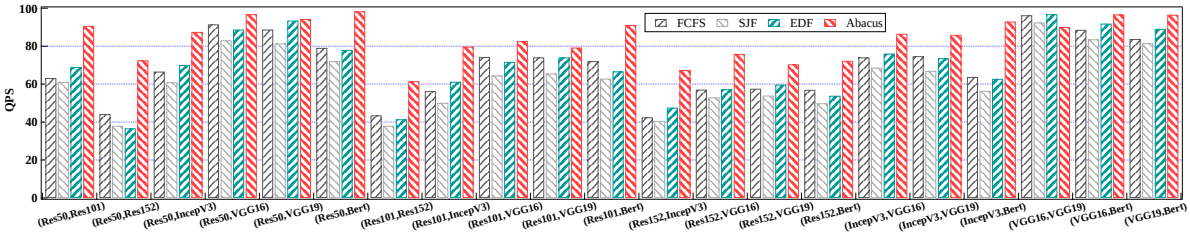

Figure 17: The peak throughput of each co-location pair with FCFS, SJF, EDF, and Abacus while guaranteeing the QoS.

kernel sizes. While these small operators cannot saturate the GPUs, FCFS, SJF, and EDF result in unnecessary queuing. On the contrary, Abacus is effective through the adaptive operator groups.

To conclude, Abacus is able to reduce the tail latencies of all pairs of DNN services and keep the QoS violation ratio at a low level. Abacus results in low latency because it has more headroom for executing more operators through scheduling operator group. In the meantime, the precise prediction of overlap-aware latency predictor guarantees the QoS with Abacus.

We can also observe that FCFS and EDF control the tail latency more effectively than SJF. However, this is achieved by dropping queries. SJF performs the worst among the four scheduling policies. This is mainly because SJF needs to predict the queries' latency ahead of the scheduling. Unlike Abacus, SJF is not able to hide the prediction overhead through pipelined scheduling proposed in Section 6.3. This also proves the effectiveness of the proposed pipelined scheduling in Abacus.

We also evaluate whether Abacus adapts to small DNNs. In this experiment, we fix the input of DNNs to be the minimum one and set the QoS target to be the 2 × solo-run latencies of the minimum input (around 10ms). Figure 16 shows the 99%-ile latencies of the small DNNs normalized to their QoS targets with Abacus. As observed, Abacus is capable of ensuring the QoS targets of small DNNs, as Abacus does not introduce extra synchronization fences (discussed in Section 6.3).

By comparing Figure 16 and Figure 14, we can find that the 99%-ile latencies of the small DNNs are closer to their QoS targets. This is because the QoS target is reduced in this experiment, and Abacus has less room to generate operator groups. In general, Abacus also works for small DNNs.

7.3 Improving Peak Throughput

In this subsection, we evaluate the effectiveness of Abacus in improving the peak serving throughput by increasing the query load to be 100QPS. In this case, the hardware is saturated, and the number of successfully processed queries per second of each scheduler is reported to be its peak throughput. The throughput presented in this experiment does not indicate the processed images per second or sequences per second (the load 100QPS is corresponding to 1500 images or sequences per second).

Figure 17 shows the peak throughput achieved with FCFS, SJF, EDF, and Abacus. As observed, Abacus improves the peak processing throughput by 25.7%, 38.1%, 25.7% on average compared with FCFS, SJF, EDF, respectively. At the same time, Abacus reduces

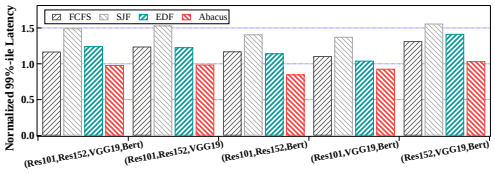

Figure 18: The 99%-ile latency in triplets- and quadruplets-wise deployments with FCFS, SJF, EDF, and Abacus.

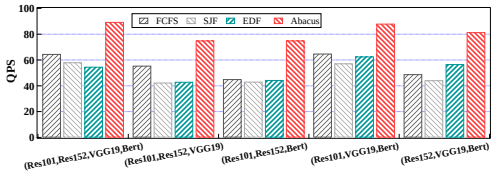

Figure 19: Peak throughputs in triplets- and quadruplets-wise deployments with FCFS, SJF, EDF, and Abacus.

55.7%, 63.2%, 54.3% QoS violations compared with FCFS, SJF, EDF in this experiment.

As observed, the throughput improvement is larger for DNN models like Resnet and Inception than VGG. For instance, Abacus achieves the largest throughput improvement in the case of (Res50, Res152). This is because the operator number of Resnet and Inception models is much larger than the operator number of VGG, and the operators of Resnet and InceptV3 can not saturate the GPU hardware. More operators and low hardware occupancy give Abacus more chance for generating operator scheduling groups.

On the contrary, we can also find that the performance of Abacus on (VGG16, VGG19) is slightly degraded with Abacus. This is because the operators of VGG are already able to saturate the GPU. In this case, there is nearly no room for operator overlap. The operators are executed in sequential, and Abacus brings in slight extra scheduling overhead.

7.4 Beyond Pair-wise Co-location

We also evaluate Abacus in triplet-wise and quadruplet-wise service deployment. Due to the large co-location possibilities, we show

Table 3: Nvidia MIGs setup

Profile Name	Fraction of Memory	Fraction of SMs
MIG 1g.5gb	1/8	1/7
MIG 2g.10gb	1/4	2/7
MIG 4g.20gb	1/2	4/7

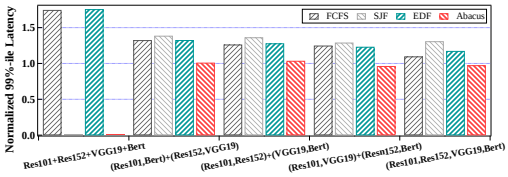

Figure 20: The 99%-ile latency of the services with MIGs.

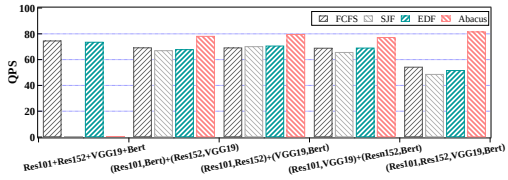

Figure 21: The peak throughputs of the services with MIGs.

the experimental results with *Res101*, *Res152*, *VGG19*, and *Bert*. Experiments with other models show similar results. In this experiment, same to the pair-wise experiment, the loads of the models are 50QPS and 100QPS when measuring the QoS and measuring the peak throughput, respectively. Figure 18 and Figure 19 show the 99%-ile latency of the benchmarks, and the peak throughputs with Abacus, FCFS, SJF, and EDF in triplet-wise and quadruplet-wise service deployment scenarios, respectively.

For triplet-wise deployment, Abacus reduces the 99%-ile latency by 21.3%, 35.3%, 20.8%, reduces the QoS violation by 87.7%, 93.4%, 85.3%, and improves the peak throughput by 51.0%, 72.3%, 57.0%, respectively compared with FCFS, SJF, and EDF. For quadruplet-wise deployment, Abacus reduces the 99%-ile latency by 16.1%, 34.3%, 21.1%, and eliminates 87.7%, 93.4%, 85.3% of the QoS violation respectively, and improves the peak supported throughput by 38.4%, 53.9%, 63.4% respectively compared with FCFS, SJF, and EDF.

We can observe that the peak throughput supported by Abacus does not reduce when the number of co-located models increase, as the scheduling overhead is well controlled. To conclude, Abacus ensures the QoS and improves the peak throughput beyond pair-wise co-location.

7.5 Integrating with MIGs

The SOTA GPU (e.g.,A100) supports MIG that divides a GPU into several isolated GPU instances [25]. The GPU instances have separate and isolated paths through the entire memory system and other resources. In this experiment, we compare the cases that directly isolate the models with MIG or co-locate them with Abacus. Three GPU instance specifications in Table 3 and four models (Res101, Res152, VGG19, Bert) are used to run the experiment.

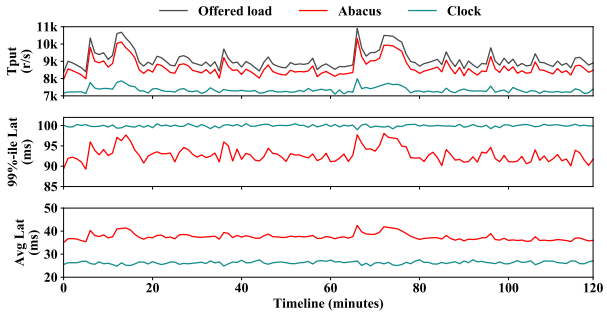

Figure 22: The throughput, 99%-ile latency, and average latency of the benchmarks with Abacus and Clockwork.

In this experiment, we compare three cases: *fully isolated*, *pair-wise isolated*, and *no isolation*. As for the full isolate case, we create four instances with *MIG 1g.5gb* and deploy each model onto a single instance. As for the pair-wise isolate case, we create two instances with *MIG 2g.10gb* and perform the pair-wise deployment. As for the no isolate case, we create one instance with *MIG 4g.20gb* and deploy the four models on this large instance.

Figure 20 and Figure 21 show the 99%-ile latency and the peak throughput in the three cases with Abacus, FCFS, SJF, and EDF. As observed from Figure 20, the 99%-ile latency of the fully-isolated case with *MIG 1g.5gb* is much longer than the QoS target. This is because DNN models that need more computation resources are not allowed to occupy some resources of other models. The queries of these models suffer from QoS violation. Instead, Abacus flexibly schedules the queries of simultaneously deployed models and eliminate such QoS violation.

We can also find that quadruplet-wise deployment in no isolation case shows similar performance with pair-wise deployment in pair-wise isolation case. If the pair-wise deployment like (VGG16, VGG19) is avoided under peak load, Abacus only needs pair-wise deployment to benefit from overlap-aware scheduling for reducing QoS violation and improving the peak throughput.

Pair-wise deployment is enough in most cases. The profiling of Abacus for training the latency predictor can be limited to pair-wise deployment with the help of MIG.

7.6 Applying in a DNN Serving Cluster

In this subsection, we integrate Abacus into Kubernetes for evaluating Abacus at the cluster level. Kubernetes itself is not aware of the interference between co-located services and results in severe QoS violation without Abacus. Therefore, we compare the method of using Kubernetes to perform the cluster-level scheduling and using Abacus to perform the node-level scheduling with STOA DNN scheduler Clockwork [17]. Clockwork performs EDF scheduling for accepted queries at the cluster level, and processes the queries in the manner of FCFS on every single GPU.

We perform the experiment on a cluster with 4 nodes (each node is equipped with four Nvidia V100 GPUs, 16 GPUs in total). Same to Clockwork [17], we replay two hours of the Microsoft Azure Functions (MAF) workload trace [43] with a QoS target of 100ms. We conduct the quadruplets-wise deployment of the four DNN

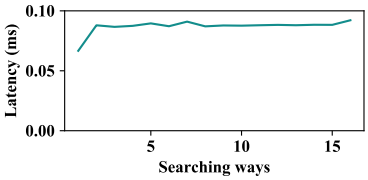

Figure 23: Duration of determining an appropriate operator group with different search ways.

models in Section 7.5 on each GPU for Abacus. Clockwork also deploys 4 instances on each GPU, but only one instance can be activated for processing DNN queries. Clockwork and Abacus use the same amount of GPU global memory.

Figure 22 shows the throughput, 99%-ile latency, and average latency of the benchmarks with Abacus and Clockwork. What should be highlighted is that Abacus supports 17.8% higher throughput than Clockwork on average, as Abacus drops much fewer queries than Clockwork. The throughput improvement mainly comes from the deterministic operator overlap. Both Abacus and Clockwork guarantee that the 99%-ile latency of the queries is shorter than the QoS target. Clockwork does not schedule some queries until it will miss the QoS deadline, leading to the 99%-ile latency being close to the QoS target. The average latency of Abacus is slightly longer than Clockwork, as Abacus trades the QoS headrooms of short queries for throughput improvement with deterministic overlap.

Moreover, because GPUs exhibit highly linear energy efficiency with respect to utilization, as stated by Kube-knots [48], Abacus also improves the energy efficiency for the large-scale GPU-based DNN serving systems.

7.7 Effectiveness of Multi-way Search

Multi-way search reduces the scheduling complexity. If we search in m-ways and the total number of DNN operators is N, the scheduling complexity is $O(log_m N)$. While we can search in parallel for the operator group to reduce the scheduling overhead, the predictor latency is an important factor.

In this experiment, we affiliate the scheduler of Abacus to a single core and evaluate the response latency of the predictor with different numbers of search ways. Figure 23 shows the time of identifying an operator group with Abacus, when different numbers of search ways are used. As observed, the latency grows from $0.066ms$ to $0.088ms$, as the search ways grow to 2. The latency does not further increase with the number of search ways.

According to our measurement, with the help of a 4-way search, most of the scheduling decisions are completed with less than three predictions, and the overall prediction latency is $0.26ms$. It is shorter than the processing time of most DNN operators, like convolution. Through pipelined scheduling, Abacus is able to hide the search time behind the execution of the current running operator group.

7.8 Overhead

In this subsection, we discuss the overhead introduced by Abacus during offline profiling and online scheduling.

Offline Profiling. Before online serving, Abacus needs offline profiling. For a pair of co-location, we sample 2,000 operator groups and run each group 100 times, done in 2 hours. We have proved that pair-wise deployment is enough in Section 7.5, and Abacus only needs to profile pair-wise co-location. Co-location like (VGG16, VGG19) can be avoided by analyzing the profiling data. If the latency of the co-located DNN models always equals that of sequential execution, Abacus does not deploy them together. In general, given N DNNs, Abacus practically divides them into several service groups of size k to resolve the scalability problem of profiling. Then only the models in the same service group are deployed together on the same GPU. The profiling complexity is reduced to $O(N)$.

Online scheduling. Abacus consumes more CPU resources than FCFS and SJF. The model of predictor occupies some main memory, which is approximately 14kB. The predictor needs a single core for fast prediction, as shown in Figure 23. Abacus does not consume extra GPU resources compared with other scheduling policies. The global memory used for storing intermediate results in the segmental model executor is small (20MB) compared with the global memory used for the whole DNN services.

7.9 Discussion and Future Work

Abacus relies on Kubernetes to perform cluster-level scheduling. An interesting future work is enhancing Abacus, so that it can automatically scale the resource usage in a DNN serving cluster. Scaling of a serving cluster includes the scaling in/out (adjust the number of the server nodes) and scaling down/up (adjust the resource of the server nodes). Based on the experiment results, Abacus can be extended to determine whether to scale out or up. At the cluster level, Abacus may also identify the DNN models that are suitable to be co-located for maximizing cluster utilization.

8 CONCLUSION

We propose Abacus that improves the system throughput while ensuring the QoS requirement of multiple DNN services deployed simultaneously on a single GPU. To achieve the above purpose, Abacus identifies the importance of predictability, and enables the precise overlap-aware latency prediction and deterministic scheduling of overlapped DNN operators for simultaneous deployment. Experimental results demonstrate the effectiveness of Abacus in handling QoS of multiple services and improving the system-wide throughput. Abacus reduces 51.3% of the QoS violation and improves the peak throughput by 29.8% on average compared with state-of-the-art solutions.

ACKNOWLEDGMENTS

This work is partially sponsored by the National Natural Science Foundation of China (NSFC) (62022057, 61832006, 61632017, 61872240, 62072297) and the Program of Shanghai Academic/Technology Research Leader under Grant 18XD1401800.

REFERENCES

[1] 2021. cgroups. https://www.kernel.org/doc/Documentation/cgroup-v2.txt.
[2] 2021. memcached. https://memcached.org.
[3] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. 2020. A multi-neural network acceleration architecture. In *ISCA*. IEEE, 940–953.

[4] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *OSDI*. 499–514.

[5] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2, 3 (2011), 1–27.

[6] Quan Chen, Zhenning Wang, Jingwen Leng, Chao Li, Wenli Zheng, and Minyi Guo. 2019. Avalon: towards qos awareness and improved utilization through multi-resource management in datacenters. In *Proceedings of the ACM International Conference on Supercomputing*. 272–283.

[7] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *ASPLOS*. 17–32.

[8] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ASPLOS* 51, 4 (2016), 681–696.

[9] Shuang Chen, Christina Delimitrou, and José F Martínez. 2019. Parties: Qos-aware resource partitioning for multiple interactive services. In *ASPLOS*. 107–120.

[10] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[11] Yujeong Choi and Minsoo Rhu. 2020. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *HPCA*. IEEE, 220–233.

[12] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E Gonzalez, Ion Stoica, and Alexey Tumanov. 2018. Inferline: Ml inference pipeline composition framework. *arXiv preprint arXiv:1812.01776* (2018).

[13] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *NSDI*. 613–627.

[14] Weihao Cui, Quan Chen, Han Zhao, Mengze Wei, Xiaoxin Tang, and Minyi Guo. 2020. E 2 bird: Enhanced Elastic Batch for Improving Responsiveness and Throughput of Deep Learning Services. *IEEE Transactions on Parallel and Distributed Systems* 32, 6 (2020), 1307–1321.

[15] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Mingyi Guo. 2019. Ebird: Elastic batch for improving responsiveness and throughput of deep learning services. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 497–505.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[17] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *OSDI*. 443–462.

[18] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse dnn models without hardware-support via tile-wise sparsity. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR*. 770–778.

[20] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[21] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *IISWC*. IEEE, 1–10.

[22] Kubernetes. 2021. Kubernetes. https://kubernetes.io.

[23] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *OSDI*. 881–897.

[24] NVIDIA. 2021. CUDA C/C++ Streams and Concurrency. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[25] Nvidia. 2021. Multi-Instance GPU. https://docs.nvidia.com/cuda/mig/index.html.

[26] NVIDIA. 2021. Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[27] NVIDIA. 2021. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[28] NVIDIA. 2021. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-Compute.

[29] NVIDIA. 2021. NVIDIA Triton Inference Server. https://github.com/NVIDIA/triton-inference-server.

[30] NVIDIA. 2021. Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[31] NVIDIA. 2021. TensorRT. https://developer.nvidia.com/tensorrt.

[32] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139* (2017).

[33] Pu Pang, Quan Chen, Deze Zeng, and Minyi Guo. 2020. Adaptive preference-aware co-location for improving resource utilization of power constrained datacenters. *IEEE Transactions on Parallel and Distributed Systems* 32, 2 (2020), 441–456.

[34] Pu Pang, Quan Chen, Deze Zeng, Chao Li, Jingwen Leng, Wenli Zheng, and Minyi Guo. 2020. Sturgeon: Preference-aware Co-location for Improving Utilization of Power Constrained Computers. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 718–727.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[36] Tirthak Patel and Devesh Tiwari. 2020. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *HPCA*. IEEE, 193–206.

[37] Heyang Qin, Syed Zawad, Yanqi Zhou, Lei Yang, Dongfang Zhao, and Feng Yan. 2019. Swift machine learning model serving scheduling: a region based reinforcement learning approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–23.

[38] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2019. MLPerf Inference Benchmark. arXiv:1911.02549 [cs.LG]

[39] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[40] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2019. INFaaS: A Model-less Inference Serving System. *arXiv preprint arXiv:1905.13348* (2019).

[41] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv:1811.02883* (2018).

[42] George AF Seber and Alan J Lee. 2012. *Linear regression analysis*. Vol. 329. John Wiley & Sons.

[43] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 205–218.

[44] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *SOSP*. 322–337.

[45] Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. 2016. Training region-based object detectors with online hard example mining. In *CVPR*. 761–769.

[46] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[47] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *CVPR*. 2818–2826.

[48] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2019. Kube-knots: Resource harvesting through dynamic container orchestration in gpu-based datacenters. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–13.

[49] Hongwei Wang, Fuzheng Zhang, Jialin Wang, Miao Zhao, Wenjie Li, Xing Xie, and Minyi Guo. 2018. Ripplenet: Propagating user preferences on the knowledge graph for recommender systems. In *CIKM*. 417–426.

[50] Mengze Wei, Wenyi Zhao, Quan Chen, Hao Dai, Jingwen Leng, Chao Li, Wenli Zheng, and Minyi Guo. 2020. Predicting and reining in application-level slowdown on spatial multitasking GPUs. *J. Parallel and Distrib. Comput.* 141 (2020), 99–114.

[51] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1049–1062.

[52] Wei Zhang, Quan Chen, Ninxing Zheng, Weihao Cui, Kaihua Fu, and Minyi Guo. 2021. Towards QoS-awareness and Improved Utilization of Spatial Multitasking GPUs. *IEEE Trans. Comput.* (2021).

[53] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. 2019. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In *ICS*. 58–68.

[54] Wei Zhang, Ningxin Zheng, Quan Chen, Yong Yang, Zhuo Song, Tao Ma, Jingwen Leng, and Minyi Guo. 2020. Ursa: Precise capacity planning and fair scheduling based on low-level statistics for public clouds. In *49th International Conference on Parallel Processing-ICPP*. 1–11.

[55] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 167–181.

[56] Han Zhao, Quan Chen, Yuxian Qiu, Ming Wu, Yao Shen, Jingwen Leng, Chao Li, and Minyi Guo. 2018. Bandwidth and Locality Aware Task-stealing for Manycore Architectures with Bandwidth-Asymmetric Memory. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 4 (2018), 1–26.

[57] Han Zhao, Weihao Cui, Quan Chen, Jingwen Leng, Kai Yu, Deze Zeng, Chao Li, and Minyi Guo. 2020. CODA: Improving Resource Utilization by Slimming and Co-locating DNN and CPU Jobs. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 853–863.

[58] Wenyi Zhao, Quan Chen, and Minyi Guo. 2018. KSM: Online Application-Level Performance Slowdown Prediction for Spatial Multitasking GPGPU. *IEEE Computer Architecture Letters* 17, 2 (2018), 187–191.

[59] Wenyi Zhao, Quan Chen, Hao Lin, Jianfeng Zhang, Jingwen Leng, Chao Li, Wenli Zheng, Li Li, and Minyi Guo. 2019. Themis: Predicting and reining in application-level slowdown on spatial multitasking GPUs. In *IPDPS*. IEEE, 653–663.

[60] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. HSM: A Hybrid Slowdown Model for Multitasking GPUs. In *ASPLOS*. 1371–1385.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Our experiments have two major parts: the evaluation of Latency and Throughput.

For all evaluations, we use seven DNN models, including ResNet50, ResNet 101, ResNet 152, Inception V3, VGG16, VGG 19, Bert. For measuring latency, we test the seven DNN models with modest serving load and record the 99%-ile latency and QoS violation ratio. For measuring throughput, we test the seven DNN models with the load that exceeds the hardware limit and record the peak supported throughput. In all experiments, we run those models according to the experiment setup on a Ubuntu Server that equips Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz and an NVIDIA Ampere 100 GPU.

Author-Created or Modified Artifacts:

```
Persistent ID:
↪ https://github.com/Raphael-Hao/Abacus/tree/master
Artifact name: Abacus

Persistent ID:
↪ https://zenodo.org/badge/latestdoi/295328892
Artifact name: Abacus
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz, A100-PCIE-40GB

Operating systems and versions: Ubuntu 20.04 running Linux kernel 5.8.0

Compilers and versions: gcc 9.3, nvcc V11.2.152

Applications and versions: ResNet50, ResNet 101, ResNet 152, Inception V3, VGG16, VGG 19, Bert

Libraries and versions: GPU Driver Version: 460.39; CUDA Version: 11.2, CUDNN Version: 8.1, pytorch 1.8.1

Key algorithms: FCFS, SJF, EDF

URL to output from scripts that gathers execution environment information.

```
https://github.com/Raphael-Hao/Abacus/blob/master/en┘
↪ vironment.txt
```